

AD-A045 415

AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO
A PRE-PROCESSOR FOR A STRUCTURED VERSION OF COBOL.(U)
MAR 77 R C HILB

F/G 9/2

UNCLASSIFIED

AFIT-CI-77-55

NL

1 of 1
ADA045415



AD A 045415

77-55

①

A PRE-PROCESSOR FOR A STRUCTURED
VERSION OF COBOL

Robert Clifford Hilb

A Thesis
Submitted to
the Graduate Faculty of
Auburn University
in Partial Fulfillment of the
Requirements for the
Degree of
Master of Science

COPY AVAILABLE TO DDC DOES NOT
REPRESENT FULLY LEGIBLE PRODUCTION

DDC
RECEIVED
OCT 20 1977
D

Auburn, Alabama

March 17, 1977

AD No.
DDC FILE COPY

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT - CI-77-55	2. GOVT ACCESSION NO.	3. REPORT'S CATALOG NUMBER Thesis
4. TITLE (and Subtitle) A Pre-Processor For A Structured Version of Cobol		5. TYPE OF REPORT & PERIOD COVERED Thesis
7. AUTHOR(s) ROBERT CLIFFORD HILB CAPTAIN, USAF		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT Student at Auburn University, Auburn, Alabama		8. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/CI Wright-Patterson AFB OH 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 11 17 Mar 77
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE March 17, 1977
		13. NUMBER OF PAGES 70 pages
		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for Public Release; Distribution Unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES JERRAL F. GUESS, Captain, USAF Director of Information, AFIT APPROVED FOR PUBLIC RELEASE AFR 190-17.		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) 012 200		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Attached		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ACCESSION FOR	
NTIS	White Section <input checked="" type="checkbox"/>
DDC	Ref. Section <input type="checkbox"/>
UNANNOUNCED	<input type="checkbox"/>
JUSTIFICATION	
BY	
DISTRIBUTION/AVAILABILITY CODES	
Dist. Avail. and/or SPECIAL	
A	23

A PRE-PROCESSOR FOR A STRUCTURED
VERSION OF COBOL

Robert Clifford Hilb

Permission is herewith granted to Auburn University to make copies of this thesis at its discretion, upon the request of individuals or institutions and at their expense. The author reserves all publication rights.

Robert C. Hilb
Signature Author

22 Feb 77
Date

Copy sent to:

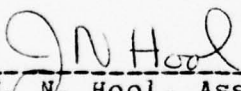
Name

Date

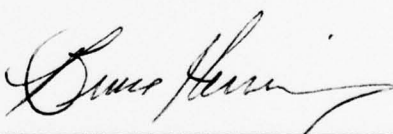
A PRE-PROCESSOR FOR A STRUCTURED
VERSION OF COBOL

Robert Clifford Hilb

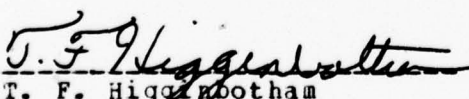
Certificate of Approval:



J. N. Hool, Associate
Professor
Industrial Engineering



B. E. Herring, Chairman
Associate Professor
Industrial Engineering



T. F. Higginbotham
Assistant Professor
Industrial Engineering



Paul F. Parks, Dean
Graduate School

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

DDC
RECEIVED
OCT 20 1977
RECEIVED
D

VITA

Robert Clifford Hilb, son of Theodore Robert and Marjory (Hayes) Hilb, was born April 21, 1948, in Plainfield, New Jersey. He attended Westfield Public Schools and graduated from Westfield Senior High School, Westfield, New Jersey in 1966. In June, 1966, he entered the United States Air Force Academy and received the degree of Bachelor of Science in Engineering Sciences (Astronautics and Computer Science) and a commission as a Second Lieutenant in the United States Air Force in June, 1970. After of a number assignments in the U. S. Air Force, he was sent by the Air Force Institute of Technology to the Graduate School, Auburn University. He married Barbara, daughter of Earl Easten and Louise (Currier) Hawkins in June, 1970.

THESIS ABSTRACT
A PRE-PROCESSOR FOR A STRUCTURED
VERSION OF COBOL

Robert Clifford Hilb

Master of Science, March 17, 1977
(B.S., U.S. Air Force Academy, 1970)

77 Typed Pages

Directed by Bruce E. Herring

A version of COBOL that permits structured programming was designed. It was implemented using a pre-processor that outputs standard COBOL. The pre-processor will indiscriminately run programs that are either structured or non-structured, or with very limited restrictions, programs with a mixture of both. A structured IF was included as a modification of the COBOL IF. In line looping was created with a redefinition of the PERFORM, maintaining all of its capabilities and adding a DO UNTIL. The PERFORM was also modified to allow a CASE construct. All new structures are completely nestable.

TABLE OF CONTENTS

LIST OF FIGURES.....	vii
I. INTRODUCTION.....	1
Statement of the Problem	
Literature Search	
Structured COBOL	
II. LANGUAGE DESIGN.....	18
IF THEN ELSE	
In-Line Looping	
III. PRE-PROCESSOR IMPLEMENTATION.....	27
IF Processing	
PERFORM Processing	
IV. CONCLUSIONS AND RECOMMENDATIONS.....	34
REFERENCES.....	36
APPENDICES.....	38
A. Source Listing of the Pre-processor.....	39
B. Example COBOL Program.....	55
C. Processed COBOL Program.....	61
D. User's Manual.....	68

LIST OF FIGURES

1. Basic Structures.....	7
2. Extended Structures.....	10
3. Top down Design.....	11
4. Program Design.....	29

I. INTRODUCTION

Structured Programming is one of the generally accepted methods of reducing software costs, both of acquisition and maintenance, while improving program quality. Until very recently the high cost of hardware relative to software resulted in the development of software that minimized the need for and, as a result, the cost of hardware. Today the situation is reversed and in the future it is predicted to become even more one-sided. With the concentration on micro-technology development, the size and cost of computer hardware has made a quantum jump downward while the available core has increased almost as much. At the same time, the size and complexity of problems that can be programmed have greatly increased. Almost everyone in the computer field, especially those in management who have seen their budgets and cost estimates for software development and maintenance soar out of sight, have realized something must be done.

According to McGowan (1975), in 1975 the U. S. Air Force estimated that 65% of its computer costs went towards software and by 1985 95% will. He estimated that overall 10 billion dollars was spent on software in 1975. Gansler

(1976) stated that U. S. Air Force avionics software development costs were \$75 per instruction while maintenance of the same software was close to \$4000 per instruction. A method that has been shown by Baker (1972a) to not only reduce initial development costs but more importantly to reduce the time and cost of debugging and increase maintainability is top down structured programming. It was Dijkstra (1968) who first formalized structured programming into a programming discipline. However, it is a method of programming that was probably unconsciously used by the best ALGOL programmers for years.

When PL/I, SIMSCRIPT and other second generation languages were developed¹, the facilities for structured programming were included, again unconsciously since the tenants of structured programming had not been formulated at the time. Of the three early languages, ALGOL had the only real facility for structured programming. Unfortunately, due to its poor implementation and the opposition and competition from a major U. S. producer of hardware and software, it was and is the least used of the three. Most programming is still done in FORTRAN or COBOL. These were languages that were originally designed to run very quickly

¹For the purposes of this thesis, FORTRAN, ALGOL, and COBOL are defined as first generation languages and the other major languages that were developed, or else redeveloped, like SIMSCRIPT was, from the mid 60's on, as second generation languages.

in small computers. They were and still are seriously lacking in ease of designability and maintainability, and as a result often have poor reliability. Due to its widespread use in academia and utter incompatibility to the methods of structured programming, many changes for FORTRAN have been recommended. Some have been implemented in both pre-processors and, for some vendors, in compilers.

COBOL has been a different story. According to Yourdon (1975c), little has been done for COBOL because of disdain for it by the academic community and the lack of research that is done by the business community in language design. However, recently there has been widespread support for changes to COBOL. In April of 1975 a conference was held by the Programming Language Committee of CODASYL¹, the Symposium on Structured Programming in COBOL-Future and Present. Many proposals were put forth by the participants, ranging from not changing COBOL at all (a minute minority) to changing it so much that it would resemble PL/I. By the end of the conference it was generally agreed that some changes were needed, although no agreement was reached on exactly what changes should be made. The changes are not close to happening. One participant, Edward Yourdon (1975b) predicted that it would be 1978 or 1979, at the earliest,

¹CODASYL, the Conference On Data Systems Language, is the body which governs the specification of the COBOL language.

before any changes would be reflected in any new versions of COBOL.

Statement of the Problem

The purpose of this study was to design a structured version of COBOL and implement it using a pre-processor. Structured programming in PL/I was used to design and implement the pre-processor. The considerations used to design the structures were: (1) as much as possible redefinitions of existing COBOL structures were used, (2) any new verbs or structures were patterned after existing COBOL structures, and (3) all existing programs could be run against the pre-processor with no changes.

Rather than drastically change the COBOL language, which would, most likely, produce resistance from experienced programmers, a gradual change would allow a more orderly transition to a fully structured COBOL. The changes made here are an addition to, rather than a change to, COBOL. This will allow the training of new programmers in structured programming while slowly transitioning the old programmers. All of the basic structured programming structures are included in this version of COBOL. As a future proposal other refinements can be added. Of course, too drastic a change to COBOL would result in a language so similar to PL/I that PL/I could be used instead. However, due to the large investment in existing COBOL programs, it will not be economically feasible, at the present, and prob-

ably not for the foreseeable future, to replace COBOL. Therefore, not only is a version of COBOL needed that will allow structured programming, but also one that will run all existing COBOL programs. When maintenance is done on existing programs the new structures should be able to be intermixed with the old. No distinction should be necessary between old and new programs; they should all run under the same job control language including the pre-processor. The route taken in this study is, then, to extend the meaning of certain COBOL verbs, while keeping all previous meanings, and adding some minor terminators and optional modifiers for clarity.

Literature Search

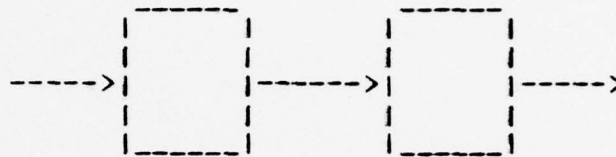
Many books and papers have been written on structured programming. In fact, it is probably the favorite subject of both programming theoreticians and practitioners at the present time. COBOL has been used almost strictly by the business community. Until recently this has resulted in the absence of research on the subject as cited before in Yourdon (1975c). The theoretical foundations for structured programming in any language were laid in 1966.

Bohm and Jacopini (1966) showed that any program with one entry and one exit (that is, an algorithmic solution to a problem as almost all programs are) could be programmed with only the following three logic structures: simple sequence, conditional branch to two choices (IF THEN ELSE),

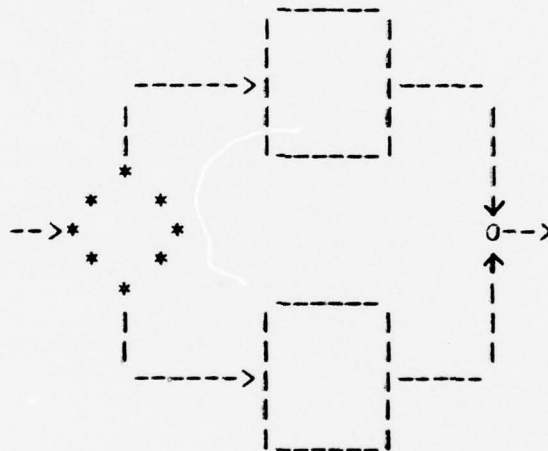
and iteration (DO WHILE) (Fig. 1). Put more simply, any program, no matter what its complexity, can be written with a combination of these three structures. Dijkstra (1968) took their basic theory and developed a practical programming methodology.

The Structured Programming presented by Dijkstra includes using only the above three structures. Any program can be developed by the appropriate nesting of these structures. The flow of program control must include single entrance and exit with no branching out of the flow allowed. Most practitioners advocate the black box or modular concept (McGowan, 1975), where each module, subroutine, function, etc. is a small easily understood section of code. Many recommend that the size be one page or less. This allows for both easy debugging and maintenance. The concept is also fundamental for top down programming. This also eliminates the forward and backward referencing of either the basic or conditional GO TO. Dijkstra (1972) maintains, and it seems obvious, that any type of branching out of the logic flow not only causes the logic of the programming to be hard to follow but is also a chief cause of error. To add to both the understandability and readability of programs, both indentation and documentation are also required. For example,

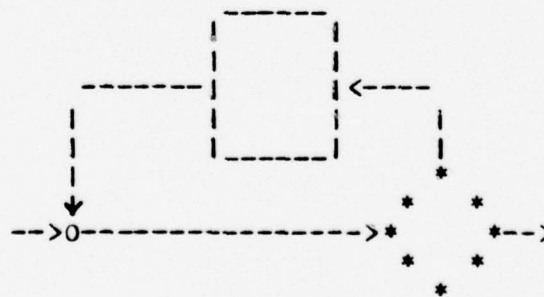
Fig. 1.--BASIC STRUCTURES



SIMPLE SEQUENCE



IF THEN ELSE



DO WHILE

PERFORM Z100 UNTIL A.

·
·
·

Z100.

IF B THEN

IF D THEN E

ELSE F

ELSE G.

is much harder to follow than

PERFORM UNTIL A.

IF B THEN

IF D THEN

E

ELSE

F

END-IF

ELSE

G

END-IF

END-PERFORM.

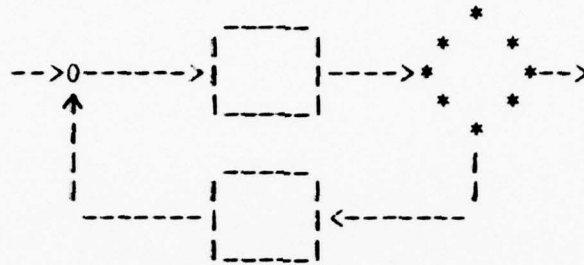
Of course for anyone else to understand, check, or change a program, it is much easier if, at the beginning of each section, is written what the programmer thinks the section will accomplish.

It has also generally been accepted (Yourdon, 1975a and McGowan, 1975) that extension to the three basic structures be allowed. These include DO CASE, DO UNTIL, and LOOP EXIT-

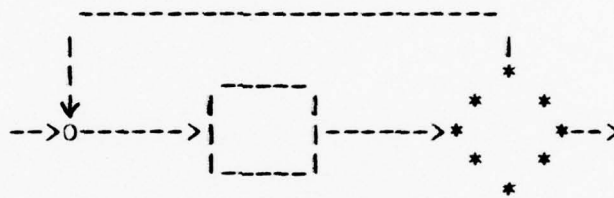
IF END-LOOP (Fig. 2). These do not affect the underlying principles of structured programming and will allow more efficient coding. Although anything can be programmed using the three basic structures, limiting programming to just the three sometimes results in awkward, inefficient, and hard-to-understand code, this is just the opposite of the purpose for structured programming. Extensions to the basic structures are sometimes necessary to avoid the "turing tarpit" (Couperus, 1975), where everything is possible but nothing is easy. Some authors still feel there should only be a limited number of structures and constructs (Yourdon, 1975c). However, one of the main tenants of structured programming is that it allows for better debugging and maintenance because it results in understandability and simplicity. What is simpler than having one construct that does what its name implies, yet replaces a group of complicated code (as many of the advanced constructs in PL/I do)?

Besides a method for coding, top down structured programming is also an overall programming strategy. Most programming has been traditionally done with a bottom up approach where the lowest level programs, subroutines, or subsystems were designed, coded, and tested first. These programs needed driver programs to test them and for integration testing with parts that other programmers had written. Problems often occurred during integration because of inconsistent data definitions and interfaces (McHenry,

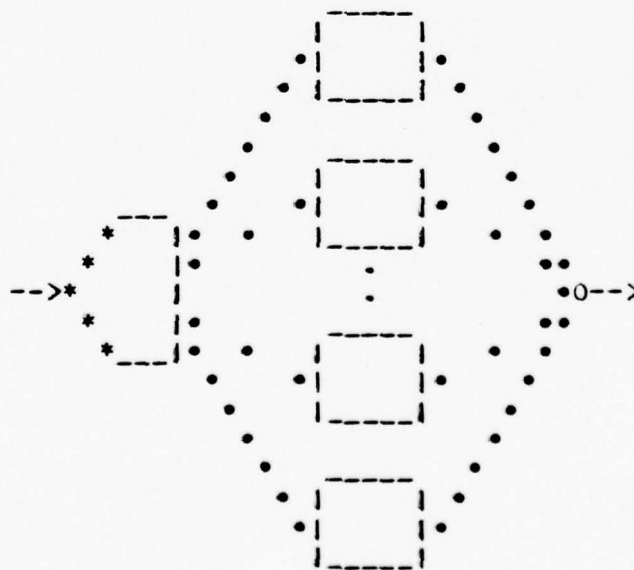
Fig. 2.--EXTENDED STRUCTURES



LOOP EXIT-IF END-LOOP



DO UNTIL

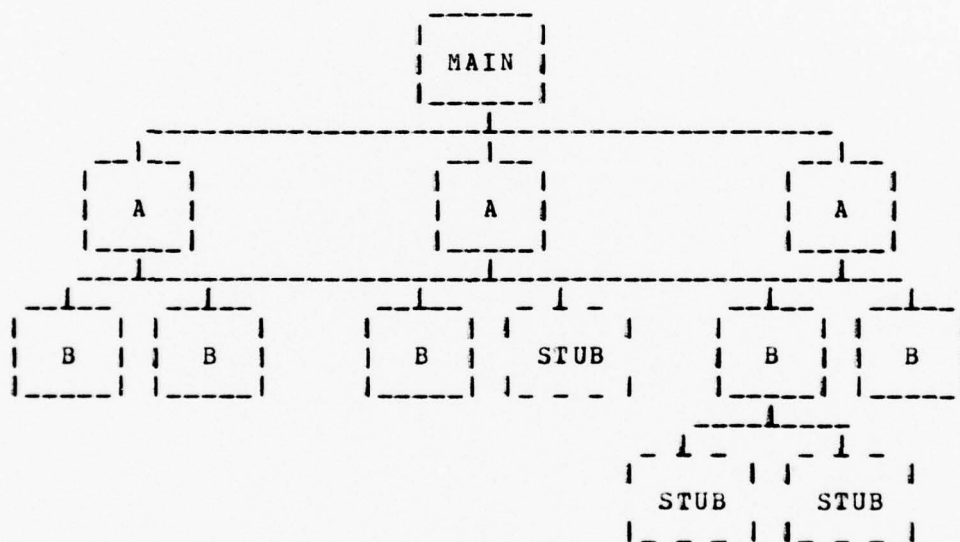


DO CASE

1973). The whole system was sometimes delayed while important segments were reworked to allow proper interfacing. The top down approach, as the name implies, is just the opposite.

Top down programming follows the natural systems approach (Yourdon, 1975a and McGowan, 1975). The system is broken down into a tree structure (Fig. 3). First the

Fig. 3.--TOP DOWN DESIGN



major functions are identified (A). From there the next level down is identified, lesser functions (B) derived from those above it. At the same time the interfaces and data definitions are defined. During this development all undefined or uncoded segments or branches of the tree are

replaced with a dummy stub. For example, if the interface is in the form of a procedure, when the procedure is called, a message or a dummy value is returned. This allows the system to be continually tested as the development proceeds. Since many errors occur in the interfacing, they are removed early in the development since they will be the most tested part of the system. Each time the system is tested all that was completed before is tested again. This is compared to the traditional bottom up method, where the overall system is not tested until the end of the development, and at that point it may be difficult to pinpoint where the problem is.

The method was first used in the now famous New York Times project (Baker, 1972a and 1972b). The project involved the automation of the New York Times morgue. It was to take the newspaper's clipping file and, via a thesaurus and abstracts, have an on-line capability of search and retrieval, with the added capability to retrieve the original article on microfiche and display it at the remote on-line terminal. The final system required more than 83,000 source lines of code. It took 132 man-months of effort. Using normal projections and the generally accepted figure of five lines of code per man-day, the project under old methods would have taken over 500 man-months. To add to this was the fact that the system had an historic low error rate, with most of these errors, as could be expected using top

down design, in the lowest level and least tested code. But these also turned out to be the easiest to fix with no error taking more than one day to correct. Equally impressive results were obtained in a second project, the mission simulation of Skylab for NASA.

Structured COBOL

All of the above literature were on the general topic of structured programming and did not address the problem of structured programming in COBOL, except for McGowan (1973) and Yourdon (1975a). They, together with McClure (1975) and numerous other authors in journals and recent publications, included small sections on how to simulate structured COBOL for those constructs in COBOL which hinder or prevent structured programming from being attained. In most cases, the authors recommend that COBOL be changed, although few give specifics on how it should be changed. Some, such as Kauffman (1975), recommend that a pre-processor be written and used until the official language changes are made and incorporated into compilers. All of the serious proposals for changes to COBOL, in regard to structured programming, were presented at the 1975 CODASYL conference.

Although some authors added a few of their own specific examples of COBOL code that hindered structured programming, almost unanimous agreement was shown for four serious shortcomings of COBOL. These are: (1) there is insufficient blocking capability, which partly results in the next

two shortcomings, (2) the IF THEN ELSE can not be fully nested and as a result does some unexpected things, (3) there is not a method for performing in-line loops, and, (4) there are no capabilities for the DO UNTIL and CASE structures. Although there was agreement on the main problems, there was none in the proposed solutions to each of the above problems. Many solutions did not fit the criteria that were previously established. Others were contradictory to what the majority felt were needed. A fourth consideration for designing the language was thus added to the original three. The structures should be those that are likely to be adopted in the official COBOL language, therefore allowing as much compatibility as possible for the future. The following articles were considered, along with those previously mentioned, when this structured version of COBOL was designed.

Benning and Nead (1975) made the broadest proposal. They proposed to use labeled BEGINS and ENDS to designate blocks in all structures. This would solve the problem of both the IF THEN ELSE and the in-line PERFORM, in the in-line PERFORM using PERFORM block... with the BEGIN-END forming the block. They also recommended a CASE structure entirely independent of the PERFORM. They suggested allowing arguments for the PERFORM and changing the AT END to an IF statement.

Couperus (1975) only suggested adding a CASE statement by allowing subscripted labels. Then either the GO TO or the PERFORM could be used to operate on the subscripted label.

Goguen (1975) recommended local terminators to solve the blocking problem. They would include the END-IF, END-PERFORM, END-KEY, END-ON, and END-AT. The END-IF would solve the problem of the nested IF THEN ELSE. She, as did Benning and Nead, suggested an in-line PERFORM, but used the END-PERFORM to delimit it. She recommended retaining the UNTIL and adding a WHILE, which would be its exact opposite, and appears a little superfluous. These two would form the DO WHILE structure. To have a DO UNTIL structure, she would add a REPEAT UNTIL at the end of the PERFORM loop. To add a CASE structure, the GO TO DEPENDING ON and the PERFORM were combined into a PERFORM DEPENDING ON... with the target a group of subscripted CASE's. The AT END, ONSIZE ERROR, and INVALID KEY would be changed by having local terminators END-AT, END-ON, and END-KEY. One final change would be making EXIT a return from anywhere in a paragraph being PERFORMed.

Hide and Croes (1975) made some PL/I type suggested changes. To solve the blocking problem, they proposed local terminators, CLOSE-IF, CLOSE-WHILE, CLOSE-LOOP, and CLOSE-PROCEDURE. Along with the CLOSE-IF, they would add an ELSEIF to solve the nested IF problem, however there does

not seem to be much advantage to adding another reserved word. In fact, they did not really specify what would be the difference between their new ELSEIF and the normal ELSE IF. It just seems harder to read. Rather than allowing an in-line PERFORM, they would add a myriad of other constructs, WHILE, REPEAT UNTIL, LOOP, DO, and PROCEDURE, each as a stand alone verb with their own individual local terminators.

Hicks (1975), as with most, would use local terminators to obtain blocking. His choice of terminators was quite unique, however. He would reverse the normal IF THEN ELSE and have IF ELSE THEN. The conditional and true part would fall between the IF and ELSE and the false part between the ELSE and THEN, with normal program flow continuing after the THEN. He would also allow the ELSEIF that Hide and Croes suggested. Instead of changing the PERFORM, he would have a general loop with the exit allowed anywhere in it. It would take the form of LOOP ... EXIT ... REPEAT. The EXIT with a conditional could be moved anywhere in the loop, so that to obtain a DO WHILE construct the EXIT would be at the top, a DO UNTIL construct would have the exit at the bottom. This seems to be a very flexible idea.

McComas (1975) would prefer blocking like Bening and Nead suggested, the BEGIN and END. However, he would not label them. He would eliminate the PERFORM altogether. He would replace the call function of it with internal proce-

dures and the expansion of CALL to include internal procedures. The looping function of PERFORM would be replaced with DO, redefining UNTIL and adding WHILE, CASE, and TIMES. To add one more PL/I type change, he would do away with the A-margin and require labels to be followed by a colon.

McGuinness (1975) would have local terminators also. The END-IF would terminate the IF and allow for complete nesting. An in-line PERFORM would be allowed with END-PERFORM terminating it. The AT END, INVALID KEY, and ONSIZE ERROR would all be changed to implied IF statements. That is, there would be an assumed IF in front of each one. This would allow the use of ELSE and END-IF after them.

Orr and Neely (1975) also recommended the local terminators END-IF and END-PERFORM to solve the blocking, nested IF, and in-line PERFORM problems. They would make CASE a separate verb and would have it in the form CASE SELECTION ... END-SELECTION. They would also like to see a feature for in-line expansion added to the COBOL language.

II. LANGUAGE DESIGN

The primary considerations for the proposed change to COBOL have been given earlier. To review the criteria, (1) as much as possible redefinitions of existing COBOL structures would be used, (2) any new verbs or structures would be patterned after existing COBOL structures, (3) all existing programs could be run against the pre-processor with no changes, and (4) the structures should be those that are likely to be adopted in the official language. A careful analysis of the problems in COBOL already presented show that the blocking problem is throughout COBOL. However, as far as structured programming is concerned, in any language, the critical structures are the IF THEN ELSE and looping. Although some of the suggestions were for a general block structure, such as the BEGIN-END, this would be a drastic change to COBOL, and except for the just mentioned two structures, the Paragraph and Section constructs already constitute at least a type of blocking. Thus local terminators were used in this design. The two selected were END-IF and END-PERFORM. The use of the END was because of the precedent in COBOL of the END DECLARATIVES. The naming of particular ENDS was used to allow better distinction of

which block was ending and to avoid ambiguity. The use of the hyphen was to avoid a possible problem when, as often occurs, the IF THEN ELSE is not followed by a period. The addition of the second IF or PERFORM, could result in an END IF IF or END PERFORM PERFORM situation, which would tend to be forgotten or to be confusing. The use of these two local terminators also solves the problems of the nested IF THEN ELSE and in-line loops. To add a DO UNTIL and CASE constructs without adding whole new constructs and local terminators, additions to the PERFORM verb were made. As it turns out, then, all changes could be made using criterion one and it was not necessary to use criterion two. Criteria three and four are discussed under the individual redefinitions.

IF THEN ELSE

The easiest redefinition to make was in solving the nested IF THEN ELSE problem. As can be seen in the Structured COBOL section of this thesis, the great majority of authors favored adding a local terminator to close the IF THEN ELSE; specifically END-IF was favored. Thus criterion four was complied with. As can be seen from the redefinition and as explained below, criterion three was also satisfied.

FORMAT			
IF condition [THEN]	{statement-1 }		{statement-2 }
	{ }	ELSE { }	{ }
	{NEXT SENTENCE}		{NEXT SENTENCE}
[END-IF]			

The restrictions that make it possible to run all old programs against the pre-processor follow. All of those that are listed in the present official COBOL language. If the optional END-IF is used, THEN becomes mandatory. Any type sentence is allowed in either the true part, between the THEN and ELSE, or the false part, between the ELSE and the END-IF. Complete nesting of IF statements can occur in either part. Of course, no periods are allowed as this would terminate the IF statement as in official COBOL. Thus all criteria for design have been met.

In-Line Looping

Although there was not as great a majority of authors who preferred the in-line PERFORM for the generalized solution to the in-line looping problem, there were a significant number who did. Also there was no consensus of those who preferred the in-line PERFORM on what form the changes should take. Therefore the simplest and most straightforward change was accomplished. To be consistent with the IF,

the END-PERFORM, as stated before, was used as the local terminator. All looping was accomplished by redefining the PERFORM formats. There are four PERFORM formats in official COBOL and the changes in all of them will be explained individually. The first format has no applicability to looping and a redefinition is really meaningless. However, for consistency, the following redefinition is made.

FORMAT 1
PERFORM [procedure-name-1 THRU procedure-name-2] [sentences END-PERFORM]

The following restrictions are added to those in official COBOL. Procedure-name-1 becomes optional; however either it or the END-PERFORM, but not both, must be present. Logically, THRU procedure-name-2 can only be used if procedure-name-1 is used. This restriction also applies to the other formats. The pre-processor will correctly process Format 1, but the result, when using it as a structured PERFORM, would be the same as a simple sequence of the same sentences, paragraphs, or sections.

The same general changes are made for the other formats but the changes become more meaningful. Format 2 becomes as follows.

FORMAT 2

```

PERFORM [procedure-name-1 THRU procedure-name-2]
    {integer-1  }
    {           } TIMES [sentences END-PERFORM]
    {identifier-1}
  
```

The only change is the addition of the in-line capability. As above THRU procedure-name-2 is only allowed when procedure-name-1 is used and when procedure-name-1 is used nothing is allowed after TIMES.

Format 3 has added to it the capability for the DO UNTIL. When UNTIL is used in COBOL, the structure that results is actually a DO WHILE since the condition is checked before entering the loop. Since the object was to minimize the changes to COBOL and a redefinition of UNTIL would both tend to confuse and would create problems in running old programs, the problem occurred about which word to use to implement the DO UNTIL. The problem has been discussed before. Tompkins (1975) points out while and until do not, by any stretch of the imagination, connote "zero-or-more" or "one-or-more". In fact, in any literal interpretation, they are logical opposites. He suggests that they should be used as inverses and that a new looping structure of REPEAT WHILE and REPEAT UNTIL be used as the "one-or-more", with the DO being the "zero-or-more". This

would be fine for a language like PL/I where a DO construct is already present. However, since this type of change is not wanted for COBOL, the word BEFORE was chosen as having the closest meaning to imply "one-or-more". Format 3 thus becomes,

FORMAT 3
<pre>PERFORM [procedure-name-1 THRU procedure-name-2] {UNTIL } { } condition-1 [sentences END-PERFORM] {BEFORE}</pre>

with no other restrictions or changes except for those in the previous two formats. Format 4 has no changes except for the addition of the in-line capability. It also has no additional restrictions or changes as shown below:

FORMAT 4
<pre>PERFORM [procedure-name-1 THRU procedure-name-2] : : [sentences END-PERFORM]</pre>

The capability that was to be added to the perform verb was the CASE construct. It did not easily fit into any of the other formats, so an additional format, Format 5, was

added. For the authors that did suggest adding a CASE construct there was no agreement on either the form of the CASE statement or the label targets. Thus the simplest form was chosen, again in the hope that this would result in compatibility with any official changes. In this case, this was the closest to compliance to criterion four that could be obtained. The resulting format is as follows.

FORMAT 5
<pre> PERFORM CASE identifier-1(arithmetic-expression) identifier-1(1) [statements identifier-1(2) statements : : identifier-1(n) statements] END-PERFORM </pre>

The subscripted identifier-1 must have consecutive subscripts from 1 to N, where N is greater than or equal to 1. The statements after the subscripted identifier-1 are optional. As a result a statement may have more than one subscript identifying it, allowing more flexibility. This is limited by the fact that the subscripts must be consecutive. Periods are optional after the arithmetic expression, after each subscripted identifier, at the end of each statement, and at the end of the construct, after END-PERFORM.

There are a few other restrictions on all of the above new constructs and redefinitions that are a result of implementation by a pre-processor. The first was mentioned for some of the individual redefinitions but applies throughout. The restriction in COBOL that no periods appear in an IF statement or else the IF statement is terminated applies equally for any of the structured constructs. This should not affect any programs and should not in any way hinder the use of any of the structured constructs. All will work with no periods in them. Programmers must be aware that when they are programming in an IF statement, that they use no period until after the END-IF. A second restriction is that no labels starting with Z are allowed. These are reserved for the pre-processor. The following reserved words are also added: END-IF, END-PERFORM, BEFORE, and CASE. The only other restriction is that a structured PERFORM not be used in a non-structured IF statement. This restriction should not matter to programmers writing a new program, since they should be using either all structured or all non-structured programming. It might affect a programmer doing maintenance on an old program. Any time a structured PERFORM is added to an old program, the programmer must insure that it is not within an IF statement or else the IF statement must also be re-written. The reason for this restriction is the pre-processor will sometimes add periods when processing a structured PERFORM. These periods are taken

care of when the structured IF is processed, so they do not affect the structured IF. The removal of these restrictions and further additions to COBOL are discussed in the conclusion of this thesis.

III. PRE-PROCESSOR IMPLEMENTATION

This chapter describes the PL/I program (Appendix A) that implements the pre-processor. PL/I was used for two reasons: one, it is a language designed for character manipulation while COBOL is not, and, two, it is a language that can be programmed using structured programming and thus would be a good example to show structured programming methods. Top down design was also used to design the program. Many changes were made between the original top down design and the final design that corresponds to the final program.

The major reason that the top down design had to be completely reworked was a result of the same problem that caused the previously discussed restrictions. The problem is that no periods are allowed in the IF statement. This meant that in the processing of the structured IF's and PERFORMS, when nesting occurred some way had to be found, in rewriting the mixture of the two, to have no periods, yet to still have proper blocking. The only fairly simple method that was found involved taking the original design, which intermixed the processing, and separating it into two job steps. Thus in essence two programs were written. This is

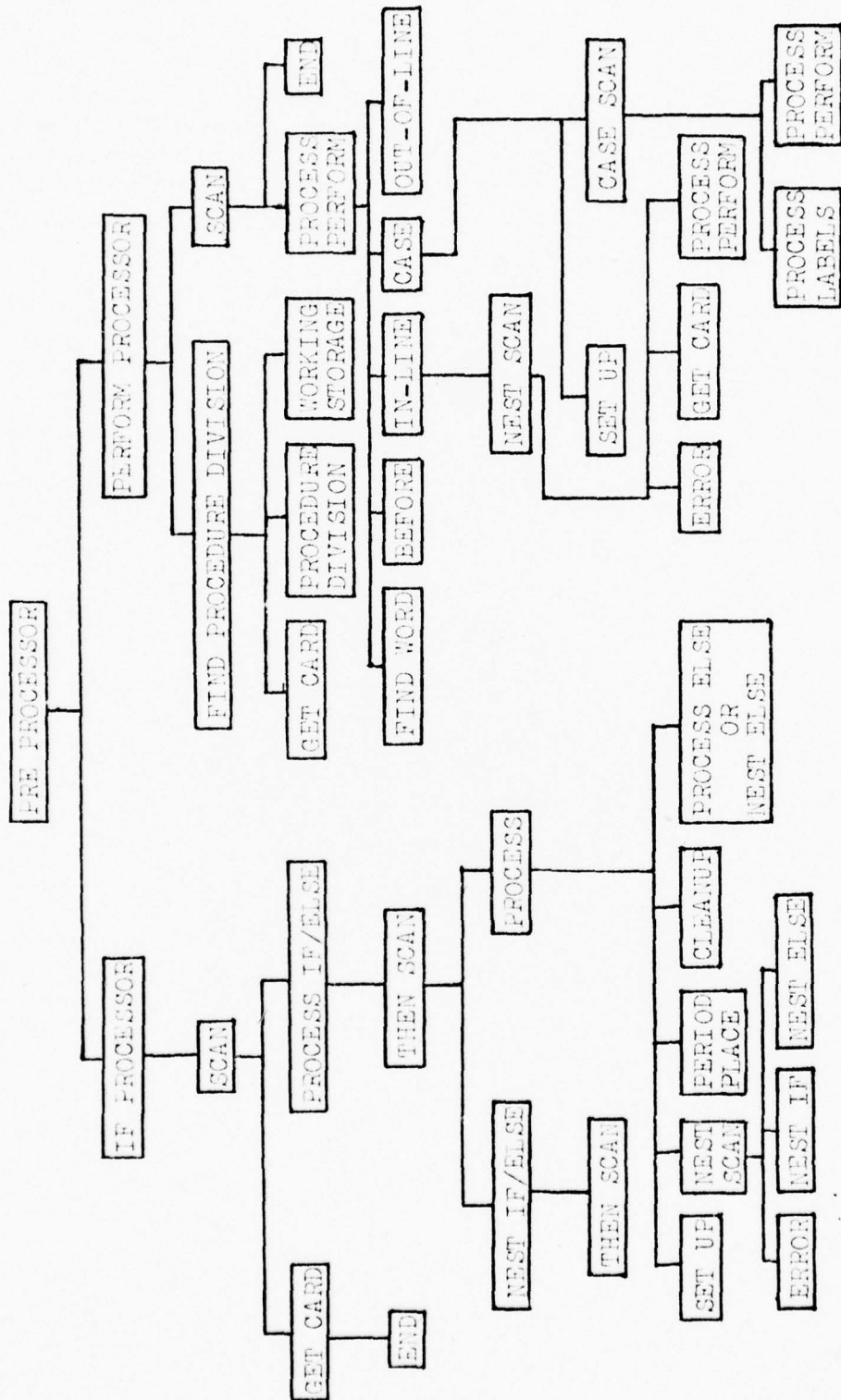
shown in Fig. 4 where an immediate breaking into two separate parts at the top of the design tree occurs. The rest of the design can be seen in Fig. 4 and the two separate job steps are explained below.

IF Processing

The first job step scans the source program for IF statements. It prints out the source listing as it scans, if requested. Each IF statement is checked. If it is a structured IF then it is processed, if not it is ignored. The general scheme is that statements between the THEN and ELSE and the ELSE and END-IF are moved to a paragraph at the end of the source program and are replaced with a PERFORM of that paragraph. Nesting can occur as many times as necessary. After this job step, all structured IF statements end up in the form: IF condition PERFORM label-Zn THRU label-Zn+1 ELSE PERFORM label-Zm THRU label-Zm+1. Since all other statements, including structured PERFORM statements, are in the PERFORMed paragraphs at the end of the source program, they can be processed in a second job step regardless of periods.

The only executable statement in the main program of the first job step is a call of SCAN. This initiates all other execution through calls on procedures. At the beginning of each procedure there is a small paragraph that explains what each procedure does. The three main procedures are called PROCESS_IF, NEST_SCAN, and NEST_IF. The

Fig. 4.--PROGRAM DESIGN



PROCESS_IF is called from the main scan any time an IF is encountered. Unless it has already been determined that the present statement is a structured IF, by the presence of a THEN in the same line, the source program is searched for a THEN or an ELSE. An IF statement that has a THEN in it is processed the same whether it is actually a structured IF or not. When an ELSE is found first, it means that the present statement is not a structured IF so a return without processing is made. The IF is transformed into the form given above and then NEST_SCAN is called to build the paragraph. NEST_SCAN builds the paragraph until either another IF is encountered or else an ELSE, END-IF, or period is encountered. If any of the latter three is found, then a return is made to PROCESS_IF, where the completion of processing is done. If an ELSE has been returned, it will be necessary to build another paragraph so PROCESS_IF is called again.

If, back in NEST_SCAN, an IF had been found, a different processing path must be taken. In this case since a paragraph is being built at the end of the source program, it is necessary to have a procedure which will consecutively build new paragraphs at the end of the string that is being built; this is NEST_IF. Its operation is almost exactly like that of PROCESS_IF except instead of building the IF-PERFORM in the middle of the source program, it does all its operations at the end of the source program. It also turns out that for simplicity and for core size savings, it is

better to dump each paragraph as it is completed, so that some paragraphs will end up ahead of the paragraphs from which they are performed. This only occurs at the end of the original program, however, and does not affect anything except readability.

PERFORM Processing

The second job step takes the output from the first job step and scans it for structured PERFORMs, processing them as encountered. Since all of the structured PERFORMs have already been removed from the structured IFs and are not allowed in the unstructured IFs, it is a simple matter to change all of the in-line PERFORMs into a PERFORM of a paragraph. The paragraph is inserted immediately after the PERFORM, using a GO TO to remove the paragraph from the stream of the program. All of the PERFORMs except for the CASE are transformed into the following form.

PERFORM Zn THRU Zn+1 conditions.

GO TO Zn+1.

Zn.

sentences.

Zn+1.

The only additional processing required is for the PERFORM BEFORE. In this case an additional simple PERFORM on the paragraph is added. This insures at least one execution of the paragraph. A large part of the job step is used to keep the labels straight in nesting situations.

Another large part of the job step is used in the processing of the PERFORM CASE. To implement the CASE, the GO TO DEPENDING ON was used. However, the target of a GO TO DEPENDING ON is very limited; only an identifier of four digits or less with a limited USAGE is allowed. To add the flexibility that is allowed by the definition of the PERFORM CASE given in the last chapter, the pre-processor adds an identifier, named CASE, in the working storage section. The target for the CASE, an arithmetic statement, is assigned to the variable CASE in a COMPUTE statement. The form that the processed CASE takes is as follows:

```

      COMPUTE CASE = (arithmetic statement).

      GO TO Zn.

case-label-1.
      statements.
      GO TO Zn+1.
case-label-2.
      :
      :
      :
case-label-n.
      statements.
      GO TO Zn+1.

Zn.
      GO TO case-label-1,
           case-label-2,
           :
           :
           .

```

case-label-n,
DEPENDING ON CASE.

Zn+1.

The liberal use of GO TOs in the processing of the PERFORMS does not result in a structured product. However, this output is invisible to the programmer. It does have the advantage of putting the processed code in the position corresponding to its position in the input, thus allowing for easier debugging. As in the first job step, all procedures have an explanation of what they do at their beginning in the program listing. PROCESS_PERFORM and NEST_SCAN have about the same purposes as the corresponding procedures in the first job step. They are not as complicated and there is no need for a procedure corresponding to NEST_IF, since there is no need to distinguish between any nested PERFORMS as they all will be put in-line in the source program.

Appendix B is a COBOL program that is written using the structures defined here. Appendix C is the same program after it has been processed. Appendix D is a user's manual for the pre-processor.

IV. CONCLUSIONS AND RECOMMENDATIONS

These changes to COBOL need to be adopted into the official COBOL language and implemented into the COBOL compilers as soon as possible. The advantages of structured programming that have been given here and shown to work in some of the projects listed, prove that there are definite cost savings and increased reliability resulting from its use. The longer the delay, the more unstructured programs will be written which will have to be maintained in the future. The major obstacle seems to be the disagreement on exactly what changes need to be made. As seen in the 1975 conference, in many cases there is wide disagreement on which structures are needed and how they should be implemented. The changes suggested here are a minimum that will allow good structured COBOL to be written. They also have the advantage of offering the least change to, and are the closest to keeping, the traditional COBOL. Thus they have the best chance of being accepted. Any change to a language should be evolutionary, both because of the investment in programs and to keep the retraining and transitioning as simple as possible. If any drastic or revolutionary changes to a language are needed, it is probably best to go to a

completely new language, thereby making a clean break with old methods and any bad habits that may have evolved.

The advantages of implementing these changes in a compiler as soon as possible are twofold; it will increase coding efficiency and it will remove those restrictions that had to be made because of the pre-processor. Since the local terminator is present in the IF, it should be possible to implement the compiler with no restrictions on the use of periods in the IF THEN ELSE. This will allow better readability and make the program easier to follow, a definite goal of structured programming.

The only other constructs whose value would make them candidates for implementation in the near future would be allowing an exit in the middle of a loop as suggested by Hicks (1975) and explained in the first chapter, and adding local terminators for the AT END, ONSIZE ERROR, and INVALID KEY as suggested by Goguen (1975). Of the two, the first would probably have the most usage and therefore should be implemented first.

This study has examined structured programming and its possible use in COBOL. Recommendations for changes to COBOL to allow structured programming have been made and implemented in a pre-processor. They definitely need to be accepted and implemented into the official language. Until this is done, the pre-processor can be used to attain structured programming in COBOL.

REFERENCES

- Baker, F. T., Chief programmer team management of production programming, IBM Systems Journal 11, 1 (1972a), pp. 56-73.
- Baker, F. T., System quality through structured programming, Proc. FJCC 1972b, pp. 339-343.
- Bening, L. C. and Nead, J. M., Some Modifications to COBOL Designed to Promote Structured Programming, Structured Programming in COBOL-future and present, New York, N. Y.: ACM, 1975, pp. 134-144.
- Bohm, C. and Jacopini, G., Flow diagrams, turing machines and languages with only two formation rules. Comm. ACM 9, 5 (May 1966), pp. 366-371.
- Couperus, J., A Proposal to Enhance the COBOL Language With a Case Construct, Structured Programming in COBOL-future and present, New York, N. Y.: ACM, 1975, pp. 15-35.
- Dahl, O., Dijkstra, E., and Hoare, C., Structured Programming, New York, N. Y.: Academic Press Inc., 1972.
- Dijkstra, E. W., Go To Statement Considered Harmful. Comm. ACM 11, 3 (Mar 1968), pp. 147-148.
- Gansler, J. S., Remarks, Managing The Development of Weapon System Software, Maxwell Air Force Base, AL.: Air University 1976, pp. 4-1 - 4-12.
- Goguen, N. H., Control Structures for Structured Programming in COBOL, Structured Programming in COBOL-future and present, New York, N. Y.: ACM, 1975, pp. 68-87.
- Hicks, J. R., Suggested Changes to COBOL to Facilitate Structured Programming, Structured Programming in COBOL-future and present, New York, N. Y.: ACM, 1975, pp. 88-94.

- Hide, D. J. and Croes, G. A., SCOBOL-Shell's Structured COBOL, Structured Programming in COBOL-future and present, New York, N. Y.: ACM, 1975, pp. 240-275.
- Kauffman, R. L., COBOL/Structured Programming (Will the Marriage Survive?), INFOSYSTEMS 22, 2 (Feb 1975), pp. 48-50.
- McClure, C. L., Structured Programming in COBOL. SIGPLAN Notices 10, 4 (Apr 1975), pp. 25-33.
- McComas, C. A., Can COBOL Be a "Structured Programming" Language? Structured Programming in COBOL-future and present, New York, N. Y.: ACM, 1975, pp. 107-114.
- McGowan, C. L., Top Down Structured Programming Techniques New York, N. Y.: Petrocelli/Charter, Inc., 1975.
- McGuinness, j., Changes to COBOL for Structured Programming, Structured Programming in COBOL-future and present, New York, N. Y.: ACM, 1975, pp. 95-106.
- McHenry, R. C., Management Concepts for Top Down Structured Programming: FSC 73-0001 Gaithersberg, Md: IBM Corporation, 1973.
- Orr, K. T. and Neely, P. M., A Modest Proposal for the adaption of COBOL to facilitate the development of structured programs, so as to make said programs beneficial to the public. Structured Programming in COBOL-future and present, New York, N. Y.: ACM, 1975, pp. 54-67.
- Tompkins, H. E., Comments on "Structured Programming in a Production Programming Environment", IEEE Trans. Software Eng. 2, 1 (Mar 1976), p. 67.
- Yourdon, E., Techniques of Program Structure and Design, Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1975a.
- Yourdon, E., Symposium on Structured Programming in COBOL. Datamation 21, 6 (Jun 1975b), p. 97.
- Yourdon, E., Teaching Structured COBOL to the Masses. Structured Programming in COBOL-future and present, New York, N. Y.: ACM, 1975c, pp. 115-133.

APPENDICES

APPENDIX A
SOURCE LISTING OF THE PRE-PROCESSOR

```
/* THIS PROGRAM TAKES A STRUCTURED VERSION OF COBOL AND
   PROCESSES IT INTO ANSI COBOL.
```

```
THE FIRST JOB STEP PROCESSES ALL IF STATEMENTS INSURING
   THAT PROPER NESTING IS OCCURING. */
```

```
MAIN:
```

```
PROCEDURE OPTIONS(MAIN) REORDER;
```

```
DECLARE FILLER CHARACTER(69) INITIAL(' ');
/* FILLER IS USED TO PAD A CHARACTER STRING WITH A PERIOD
   AND BLANKS WHERE NECESSARY. */
```

```
DECLARE ENDING CHARACTER(1);
/* ENDING IS A PLACE TO PUT A BLANK OR PERIOD. */
```

```
DECLARE CARD CHARACTER(160) VARYING INITIAL((160)' ');
/* CARD HOLDS THE INPUT CARD IMAGE PADDED WITH BLANKS. */
```

```
DECLARE BLANKS CHARACTER(160) INITIAL((160)' ');
```

```
DECLARE COUNT PICTURE '999' INITIAL(500);
/* COUNT HOLDS THE CURRENT LABEL COUNT. */
```

```
DECLARE STARS CHARACTER(80) INITIAL((80)'*');
```

```
ON STRINGRANGE PUT DATA;
```

```
OPEN FILE(ERRORS) PRINT;
```

```
/* ERROR MESSAGES ARE PRINTED OUT ON ERRORS. */
```

```
/* OTHER FILES USED: OUTPUT IS WHERE THE NON-
   NESTED SOURCE IS STORED,
   BOTTOM IS WHERE ALL NESTED
   SOURCE IS PLACED, IT IS CONCATENATED TO
   OUTPUT IN THE NEXT JOB STEP. */
```

```
/* THE CALL ON SCAN IS THE ONLY STATEMENT EXECUTED IN THE
   MAIN PROGRAM. IT INITIATES ALL OTHER ACTION IN
   THE FORM OF PROCEDURE CALLS. */
```

```
CALL SCAN;
```

```
/******
```

```
GET_CARD:
```

```
/* HANDLES ALL INPUT AND OUTPUT WHEN WE ARE NOT IN A NESTING
   SITUATION. REMOVES SEQUENCE NUMBERS AND ANYTHING IN
   COLUMNS 73-80. THEN PADS WITH BLANKS. */
```

```
PROCEDURE;
```



```

ON ENDFILE(SYSIN) CALL CLEANUP;
PUT FILE(OUTPUT) EDIT(SUBSTR(CARD,1,80)) (COL(1),A(80));
GET EDIT(CARD) (COL(1),A(80));
PUT EDIT(CARD) (COL(1),A(80));
CARD = ' ' || SUBSTR(CARD,7,66) || BLANKS;

```

```

END GET_CARD;

```

```

/*****/

```

```

SCAN:

```

```

/* SCAN IS IN PROGRESS WHENEVER THE PROGRAM IS NOT IN AN
   IF NEST. SCANS FOR IF STATEMENTS, CALLS FOR
   PROCESSING WHEN FOUND. */

```

```

PROCEDURE;

```

```

DO WHILE ('1'B);
  IF INDEX(CARD,' IF ') > 0 THEN
    IF INDEX(CARD,' THEN ') > 0 THEN
      CALL PROCESS_IF(COUNT,'THEN','0'B);
    ELSE CALL PROCESS_IF(COUNT,'THEN','1'B);
  CALL GET_CARD;
END;
END SCAN;

```

```

/*****/

```

```

ERROR:

```

```

/* ERROR PRINTS OUT AN APPROPRIATE ERROR MESSAGE AND
   TERMINATES THE PRE-PROCESSOR. HERE IS WHERE A
   METHOD FOR PREVENTING FURTHER JOB STEPS FROM
   EXECUTING COULD BE PLACED. */

```

```

PROCEDURE(MESSAGE);

```

```

DECLARE MESSAGE CHARACTER(80) VARYING;

```

```

PUT FILE(ERRORS) EDIT(STARS) (A(80));
PUT SKIP(2) FILE(ERRORS) EDIT(MESSAGE) (COL(1),A(80));
PUT SKIP(2) FILE(ERRORS) EDIT(STARS) (A(80));
CALL CLEANUP;

```

```

END ERROR;

```

```

/*****/

```

```

CLEANUP: /* A CONVENIENT PLACE TO STOP. */

```

```

PROCEDURE;

```

```
STOP;
END CLEANUP;
```

```
/******
```

```
PERIOD_PLACE:
```

```
/*  PROCEDURE FINDS THE END OF A SENTENCE OR PARAGRAPH
    AND PLACES A PERIOD THERE OR REMOVES IT DEPENDING
    ON PLACE. */
```

```
PROCEDURE (PLACE, STRINGS);
```

```
DECLARE STRINGS CHARACTER(6000) VARYING;
DECLARE PLACE BIT(1);
DECLARE (I, J, K) FIXED BINARY(15) INITIAL(0);
```

```
I = VERIFY(STRINGS, ' ');
```

```
/*  ALTERNATE BETWEEN FINDING THE NEXT BLANK AND THE NEXT
    NON-BLANK UNTIL ONLY BLANKS REMAIN. */
```

```
DO WHILE (I > 0);
  J = INDEX(SUBSTR(STRINGS, K), ' ') + K;
  I = VERIFY(SUBSTR(STRINGS, J), ' ');
  K = K + I;
END;
```

```
/*  CHECK IF THE LAST CHARACTER IS ALREADY A PERIOD. */
IF PLACE THEN DO;
  IF SUBSTR(STRINGS, J-2, 1) = '.' THEN
    SUBSTR(STRINGS, J-1, 1) = '-';
END; ELSE
  IF SUBSTR(STRINGS, J-2, 1) = '.' THEN
    SUBSTR(STRINGS, J-2, 1) = ' ';
```

```
END PERIOD_PLACE;
```

```
/******
```

```
PROCESS_IF:
```

```
/*  THIS PROCEDURE WILL PROCESS THE OUTER MOST NEST OF AN
    IF STATEMENT. */
```

```
PROCEDURE (LABEL_#, KEY_WORD, SET) RECURSIVE;
```

```
DECLARE SET BIT(1);
```

```
/*  SET, DO WE KNOW ALREADY ITS A STRUCTURED IF? */
```

```
DECLARE KEY_WORD CHARACTER(6) VARYING;
```

```
/*  KEY_WORD, ARE WE AT AN ELSE OR AN END-IF? */
```

```

DECLARE SENTENCE CHARACTER(6000) VARYING INITIAL('');
/* STORAGE FOR CODE */

DECLARE (LABEL_#,TEMP_LABEL_#) PICTURE '999';
/* STORAGE FOR DIFFERENT LABEL COUNTS. */

/* NEXT LOOP SCANS FOR EITHER A THEN OR AN ELSE.
   IF THEN IS FOUND THEN WE COULD HAVE A STRUCTURED
   IF AND PROCESSING CONTINUES. IF AN ELSE IS FOUND
   WE DO NOT HAVE A STRUCTURED IF AND WE RETURN. */

DO WHILE(SET);
  CALL GET_CARD;
  IF INDEX(CARD,' IF ')>0 & INDEX(CARD,' THEN ')>0 THEN
    CALL NEST_IF(LABEL_#,'THEN',SENTENCE,'0'B);
  IF INDEX(CARD,' THEN ')>0 THEN SET='0'B;
  IF (INDEX(CARD,' ELSE ')>0 | INDEX(CARD,'. ')>0
    & INDEX(CARD,' THEN ')>0 THEN DO;
    /* NON-STRUCTURED IF, ALL PERIODS THAT HAVE JUST
       BEEN PLACED MUST BE REMOVED. */
    IF INDEX(CARD,' ELSE ')>0 & INDEX(SENTENCE,'. ')>0 THEN
      CALL PERIOD_PLACE('0'B,SENTENCE);
    RETURN;
  END;
END;

I = INDEX(CARD,KEY_WORD)+4;
PUT FILE(OUTPUT) EDIT(SUBSTR(CARD,1,I)) (COL(1),A(80));
SENTENCE = SENTENCE || '          Z' || LABEL_# || FILLER;
SENTENCE = SENTENCE || '          ' || SUBSTR(CARD,I,68);

TEMP_LABEL_# = LABEL_# + 1;

/* WE NOW CAN PLACE A PERIOD IF WE KNOW WE ARE
   COMPLETELY OUT OF THE NESTING. */
IF KEY_WORD='ELSE' THEN
  ENDING = '.';
ELSE
  ENDING = ' ';
CARD = '          PERFORM Z' || LABEL_# || ' THRU Z'
      || TEMP_LABEL_# || ENDING;
PUT FILE(OUTPUT) EDIT(CARD) (COL(1),A(80));

LABEL_# = LABEL_# + 2;

/* AT THIS POINT ALL IS SET UP TO NEST THE IF SO WE CALL
   NEST_SCAN TO FIND EITHER ELSE, END-IF, OR A PERIOD. */

CALL NEST_SCAN(KEY_WORD,LABEL_#,SENTENCE);
CALL PERIOD_PLACE('1'B,SENTENCE);

```

```

SENTENCE = SENTENCE || '          Z' || TEMP_LABEL_# || FILLER;
PUT FILE (BOTTOM) EDIT (SENTENCE) (A);
SENTENCE = '';

```

```

/* IF WE RETURNED WITH ELSE WE MUST PROCESS THE FALSE PART
   OF THE STRUCTURE, OTHERWISE WE ARE THROUGH. */

```

```

IF KEY_WORD='ELSE' THEN
  CALL PROCESS_IF (LABEL_#, 'ELSE', '0'B);

```

```

END PROCESS_IF;

```

```

/*****

```

```

NEST_SCAN:

```

```

/* NEST_SCAN STORES ALL CARD IMAGES UNTIL IT FINDS A KEY
   WORD, IT THEN RETURNS WITH THE STORED IMAGES. */

```

```

PROCEDURE (WORD, COUNTER, LIST) RECURSIVE;

```

```

DECLARE WORD CHARACTER (6) VARYING;
/* RETURNS WITH KEY WORD. */

```

```

DECLARE COUNTER PICTURE '999';

```

```

DECLARE LIST CHARACTER (6000) VARYING;

```

```

ON ENDFILE (SYSIN) CALL ERROR ('STRUCTURE CLOSING MISSING');

```

```

/* SCAN UNTIL IF, ELSE, END-IF, OR A PERIOD IS FOUND. */

```

```

DO WHILE ('1'B);

```

```

  GET EDIT (CARD) (COL (1), A (80));
  PUT EDIT (CARD) (COL (1), A (80));
  CARD = '          ' || SUBSTR (CARD, 7, 66) || BLANKS;

```

```

  IF INDEX (CARD, ' ELSE ') > 0 THEN DO;
    WORD = 'ELSE';
    RETURN;
  END;

```

```

/* IF WE FIND AN IF, WE START NESTING. */
IF INDEX (CARD, ' IF ') > 0 THEN DO;
  IF INDEX (CARD, ' THEN ') > 0 THEN
    CALL NEST_IF (COUNTER, 'THEN', LIST, '0'B);
  ELSE CALL NEST_IF (COUNTER, 'THEN', LIST, '1'B);
END;

```

```

/* END-IF MUST BE REMOVED, PROCESSING FOR AN
   END-IF OR A PERIOD IS THE SAME SO THE

```

```

        SAME KEY WORD CAN BE RETURNED.
I = INDEX(CARD,'END-IF');
J = INDEX(CARD,'. ');
IF I>0 | J>0 THEN DO;
    WORD = 'END-IF';
    IF I>0 THEN
        SUBSTR(CARD,I,6) = ' ';
    RETURN;
END;

/* NO KEY WORDS FOUND SO STORE CARD IMAGE. */

LIST = LIST || CARD;

END;

END NEST_SCAN;

/*****/

NEST_IF:

/* NEST_IF IS ALMOST A COPY OF PROCESS_IF EXCEPT INSTEAD
   OF PRINTING OUT CARDS IN THE STREAM OF THE PROGRAM
   THEY ARE PUT AT THE END OF THE SOURCE PROGRAM. */

PROCEDURE(NUMBER,TYPE,LISTS,SET1) RECURSIVE;

DECLARE SET1 BIT(1);
DECLARE (NUMBER,NUM) PICTURE '999';
DECLARE (LISTS,TEMP) CHARACTER(6000) VARYING;
DECLARE TYPE CHARACTER(6) VARYING;
DECLARE STRING CHARACTER(80) VARYING INITIAL('');
DECLARE POINTER FIXED BINARY(15);

POINTER = LENGTH(LISTS);

DO WHILE(SET1);
    LISTS = LISTS || SUBSTR(CARD,1,80);
    GET EDIT(CARD) (COL(1),A(80));
    PUT EDIT(CARD) (COL(1),A(80));
    CARD = ' ' || SUBSTR(CARD,7,66) || BLANKS;
    IF INDEX(CARD,' IF ')>0 & INDEX(CARD,' THEN ')>0 THEN
        CALL NEST_IF(NUMBER,'THEN',LISTS,'0'B);
    IF INDEX(CARD,' THEN ')>0 THEN SET1='0'B;
    IF (INDEX(CARD,' ELSE ')>0 | INDEX(CARD,'. ')>0)
        & INDEX(CARD,' THEN ')=0 THEN DO;
        I = INDEX(SUBSTR(LISTS,POINTER),' ');
        DO WHILE (I>0);
            SUBSTR(LISTS,I+POINTER-1,1) = ' ';
            I = INDEX(SUBSTR(LISTS,POINTER),' ');
        END;
    END;

```



```

        RETURN;
    END;
END;

NUM = NUMBER + 1;
I = INDEX(CARD,TYPE)+4;
LISTS = LISTS || SUBSTR(SUBSTR(CARD,1,I)||BLANKS,1,80);
TEMP = '          Z' || NUMBER || FILLER;
TEMP = TEMP || SUBSTR(CARD,I,80);

IF INDEX(CARD,'. ')>0 | TYPE='ELSE' THEN
    ENDING = '. '; ELSE ENDING = ' ';
STRING = '          PERFORM Z' || NUMBER || ' THRU Z'
        || NUM || ENDING || BLANKS;
LISTS = LISTS || STRING;

NUMBER = NUMBER + 2;

CALL NEST_SCAN(TYPE,NUMBER,TEMP);
CALL PERIOD_PLACE('1'B,TEMP);

TEMP = TEMP || '          Z' || NUM || FILLER;
PUT FILE(BOTTOM) EDIT(TEMP) (A);

IF TYPE='ELSE' THEN DO;
    CALL NEST_IF(NUMBER,'ELSE',LISTS,'0'B);
END;
CALL PERIOD_PLACE('1'B,LISTS);

END NEST_IF;

END MAIN;

```

```

/* THIS SECOND JOB STEP TAKES THE OUTPUT FROM THE PREVIOUS
STEP WITH ALL NESTED IF'S PROCESSED AND PROCESSES IT
FOR IN-LINE PERFORMS. */

```

```

MAIN:

```

```

  PROCEDURE OPTIONS (MAIN) REORDER;

```

```

  DECLARE (CARD, NEW_CARD) CHARACTER(80) VARYING INITIAL((80)' ');
  /* CARD IMAGE STORAGE */

```

```

  DECLARE (COUNT, COUNT_PLUS) PICTURE '999' INITIAL(100);
  /* LABEL COUNT STORAGE */

```

```

  DECLARE NEXT_WORD CHARACTER(40) VARYING;
  DECLARE CASE_LABEL CHARACTER(40) VARYING;
  DECLARE CASE_NUMBER PICTURE '9' INITIAL(1);
  DECLARE STARS CHARACTER(80) INITIAL((80)'*');
  DECLARE RESERVED(30) CHARACTER(10) VARYING INITIAL
    (' ACCEPT ',' ADD ',' ALTER ',' APPLY ',' CALL ',' CANCEL ',
     ' CLOSE ',' COMPUTE ',' COPY ',' DISPLAY ',' DIVIDE ',
     ' ENTER ',' ENTRY ',' EXAMINE ',' GO ',' GOBACK ',' IF ',
     ' MOVE ',' MULTIPLY ',' OPEN ',' PERFORM ',' READ ',
     ' REWRITE ',' SEEK ',' START ',' STOP ',' SUBTRACT ',
     ' TRANSFORM ',' USE ',' WRITE ');

```

```

  OPEN FILE(ERRORS) PRINT;

```

```

  CALL FIND_PROCEDURE_DIVISION;
  CALL SCAN;

```

```

/*****

```

```

GET_CARD:

```

```

  /* HANDLES ALL CARD IMAGES WHEN NOT IN NESTING SITUATION. */

```

```

  PROCEDURE;

```

```

  PUT FILE(OUT) EDIT(CARD) (COL(1), A(80));
  GET EDIT(CARD) (COL(1), A(80));
  PUT EDIT(CARD) (COL(1), A(80));

```

```

END GET_CARD;

```

```

/*****

```

```

FIND_PROCEDURE_DIVISION:

```

```

  /* THIS PROCEDURE SCANS FOR THE WORKING STORAGE SECTION
  IF IT DOES NOT FIND ONE IT MAKES ONE.
  IN EITHER CASE IT ADDS THE RESERVED WORD CASE

```

FOR THE PROCESSING OF CASE STATEMENTS. */

PROCEDURE;

DECLARE (I,J) BIT(1) INITIAL('1'B);
DECLARE K PICTURE '9';

DO WHILE (INDEX(CARD,'PROCEDURE DIVISION.')=0);

CALL GET_CARD;

/* IF THE PROCEDURE DIVISION IS FOUND FIRST, THEN A WORKING
STORAGE SECTION MUST BE ADDED. */

IF INDEX(CARD,'PROCEDURE DIVISION.')>0 THEN DO;

IF I THEN DO;

PUT FILE(OUT) EDIT(' WORKING-STORAGE SECTION.')
(COL(1),A(80));

DO K=1 TO 9;

PUT FILE(OUT) EDIT(' 77 CASE' || K ||
' PICTURE 9(4).') (COL(1),A(80));

END;

I = '0'B;

J = '0'B;

END; ELSE J = '0'B;

END;

/* IF WORKING STORAGE IS FOUND FIRST NEED ONLY TO ADD CASE. */

IF INDEX(CARD,'WORKING-STORAGE ')>0 THEN DO;

PUT FILE(OUT) EDIT(CARD) (COL(1),A(80));

DO K=1 TO 8;

PUT FILE(OUT) EDIT(' 77 CASE' || K ||
' PICTURE 9(4).') (COL(1),A(80));

END;

CARD = ' 77 CASE9 PICTURE 9(4).';

I = '0'B;

END;

END;

END FIND_PROCEDURE_DIVISION;

/*****/

SCAN:

PROCEDURE;

/* EVERYTHING STARTS WITH A PERFORM. */

ON ENDFILE(SYSIN) CALL CLEANUP;

DO WHILE('1'B);

IF INDEX(CARD,'PERFORM')>0 THEN

CALL PROCESS_PERFORM(COUNT,CASE_NUMBER);

CASE_NUMBER = 1;

```

        CALL GET_CARD;
    END;
END SCAN;

/*****/

NEST_SCAN:

    /*  ONCE WE ARE IN A PERFORM PARAGRAPH SEARCH FOR AN
        END-PERFORM OR A NESTED PERFORM.  */

    PROCEDURE(KEY_WORD) RECURSIVE;

    ON ENDFILE(SYSIN) CALL ERROR('STRUCTURE CLOSING MISSING');

    DO WHILE ('1'B);

        IF INDEX(CARD,'PERFORM')>0 THEN DO;
            COUNT = COUNT + 1;
            CALL PROCESS_PERFORM(COUNT,CASE_NUMBER);
        END;

        IF INDEX(CARD,'END-PERFORM')>0 THEN DO;
            KEY_WORD = 1;
            RETURN;
        END;

        CALL GET_CARD;

    END;
END NEST_SCAN;

/*****/

FIND_WORD:

    /*  SEARCHES FOR NEXT WORD IN CARD AND RETURNS IT.
        CHANGES NEW_CARD TO THE REMAINDER OF CARD.  */

    PROCEDURE(WORD);

    DECLARE WORD CHARACTER(40) VARYING;

    NEW_CARD = SUBSTR(CARD,INDEX(CARD,WORD));
    NEW_CARD = SUBSTR(NEW_CARD,INDEX(NEW_CARD,' '));
    NEW_CARD = SUBSTR(NEW_CARD,VERIFY(NEW_CARD,' '));
    NEXT_WORD = SUBSTR(NEW_CARD,1,INDEX(NEW_CARD,' ')-1);

    END;

/*****/

```

ERROR:

PROCEDURE (MESSAGE);

DECLARE MESSAGE CHARACTER (80) VARYING;

PUT FILE (ERRORS) EDIT (STARS) (A (80));

PUT SKIP (2) FILE (ERRORS) EDIT (MESSAGE) (COL (1), A (80));

PUT SKIP (2) FILE (ERRORS) EDIT (STARS) (A (80));

CALL CLEANUP;

END ERROR;

/* **** */

CLEANUP:

PROCEDURE;

STOP;

END CLEANUP;

/* **** */

CONDITIONAL_SCAN:

PROCEDURE;

ON ENDFILE (SYSIN) CALL ERROR ('IMPROPER CONDITIONAL');

DO WHILE ('1'B);

IF INDEX (NEW_CARD, ' ', '.', ') > 0 THEN RETURN;

DO I = 1 TO 30;

IF INDEX (NEW_CARD, RESERVED (I)) > 0 THEN RETURN;

END;

CALL GET_CARD;

NEW_CARD = CARD;

END;

END CONDITIONAL_SCAN;

/* **** */

PROCESS_PERFORM:

/* WHEN PROCESSING THE PERFORM, WE TREAT BEFORE AND CASE
DIFFERENTLY THEN OTHER IN LINE PERFORMS AND DO
NOTHING WITH OUT OF LINE PERFORMS. */

PROCEDURE (COUNT, CASE_NUMBER) RECURSIVE;

DECLARE (COUNT, TEMP) PICTURE '999';

DECLARE (CASE_NUMBER, OLD_CASE) PICTURE '9';

DECLARE KEY_WORD FIXED BINARY (15) INITIAL (0);


```

CALL FIND_WORD('PERFORM');
COUNT_PLUS = COUNT + 1;

/* BEFORE--PERFORM ONCE AND THEN CHANGE TO AN UNTIL. */
IF NEXT_WORD='BEFORE' THEN DO;
  SUBSTR(NEW_CARD,INDEX(NEW_CARD,'BEFORE'),6) = 'UNTIL';
  PUT FILE(OUT) EDIT('          PERFORM Z' || COUNT ||
    ' THRU Z' || COUNT_PLUS || '. '
    ) (COL(1),A(80));
  NEXT_WORD = 'UNTIL';
END;

/* CHANGE THE IN LINE PERFORM TO AN OUT OF LINE ONE. */
IF NEXT_WORD='UNTIL' | NEXT_WORD='VARYING' THEN DO;

  PUT FILE(OUT) EDIT('          PERFORM Z' || COUNT ||
    ' THRU Z' || COUNT_PLUS || ' ' ||
    NEW_CARD) (COL(1),A(80));
  CARD = '';

/* SCAN TO FIND THE END OF THE CONDITIONAL. */
IF INDEX(NEW_CARD||' ','.-')=0 THEN
  CALL CONDITIONAL_SCAN;

  PUT FILE(OUT) EDIT('          GO TO Z' || COUNT_PLUS ||
    '. ') (COL(1),A(80));
  PUT FILE(OUT) EDIT('          Z' || COUNT || '. ')
    (COL(1),A(80));
  COUNT = COUNT + 1;
  TEMP = COUNT;

  CALL NEST_SCAN(KEY_WORD);

  IF KEY_WORD=1 THEN DO;
    PUT FILE(OUT) EDIT('          GO TO Z' || TEMP ||
      '. ') (COL(1),A(80));
    CARD = '          Z' || TEMP || '. ';
  END; ELSE
    CALL ERROR('IMPROPER STRUCTURE CLOSING');
  COUNT = COUNT + 1;
  KEY_WORD = 0;
END;

IF NEXT_WORD='CASE' THEN DO;

/* ISOLATE THE CASE LABEL, ASSIGN TARGET TO CASE. */
CALL FIND_WORD('CASE');
I = INDEX(NEW_CARD,'(');
CASE_LABEL=SUBSTR(NEW_CARD,1,I-1);
PUT FILE(OUT) EDIT('          COMPUTE CASE' ||
  CASE_NUMBER || ' = ' ||
  SUBSTR(NEW_CARD,I,INDEX(NEW_CARD,')')-I+1)

```

```

      || '. ') (COL(1),A(80));
    OLD_CASE = CASE_NUMBER;
    CASE_NUMBER = CASE_NUMBER + 1;
    CARD = '          GO TO Z' || COUNT || '. ';
    COUNT = COUNT + 1;

```

```

    CALL CASE_SCAN(OLD_CASE);

```

```

  END;

```

```

  CALL PROCESS_TIMES;
  CALL PROCESS_BEFORE;

```

```

END PROCESS_PERFORM;

```

```

/*****

```

```

PROCESS_BEFORE:

```

```

/*  PROCESS_BEFORE LOOKS FOR NON STRUCTURED BEFORES
    AND THEN PROCESSES THEM IF FOUND      */

```

```

  PROCEDURE;

```

```

  DO WHILE('1'B);
    I = INDEX(CARD,' BEFORE ');
    IF I>0 THEN DO;
      PUT FILE(OUT) EDIT(SUBSTR(CARD,1,I)) (COL(1),A(80));
      SUBSTR(CARD,I,8) = ' UNTIL ';
      SUBSTR(CARD,1,INDEX(CARD,'PERFORM')-1) = ' ';
      RETURN;
    END;
    IF INDEX(NEW_CARD||' ',' '.>0 THEN RETURN;
    DO I=1 TO 30;
      IF INDEX(NEW_CARD,RESERVED(I))>0 THEN RETURN;
    END;
    CALL GET_CARD;
    NEW_CARD = CARD;
  END;

```

```

END PROCESS_BEFORE;

```

```

/*****

```

```

PROCESS_TIMES:

```

```

/*  PROCESS_TIMES LOOKS FOR STRUCTURED TIMES
    AND THEN PROCESSES THEM IF FOUND      */

```

```

  PROCEDURE;
  CALL FIND_WORD(NEXT_WORD);
  IF NEXT_WORD='TIMES' THEN DO;

```

```

CALL FIND_WORD(' PERFORM ');
PUT FILE(OUT) EDIT('                PERFORM Z' || COUNT ||
' THRU Z' || COUNT_PLUS || ' ' ||
NEW_CARD) (COL(1),A(80));
CARD = '';
PUT FILE(OUT) EDIT('                GO TO Z' || COUNT_PLUS
|| ' ') (COL(1),A(80));
PUT FILE(OUT) EDIT('                Z' || COUNT || ' ')
(COL(1),A(80));
COUNT = COUNT + 1;
TEMP = COUNT;
CALL NEST_SCAN(KEY_WORD);
IF KEY_WORD=1 THEN DO;
    PUT FILE(OUT) EDIT('                GO TO Z' ||
TEMP || ' ') (COL(1),A(80));
    CARD = '                Z' || TEMP || ' ';
END; ELSE
    CALL ERROR('IMPROPER STRUCTURE CLOSING');
COUNT = COUNT + 1;
KEY_WORD = 0;
END;

```

```
END PROCESS_TIMES;
```

```
/******
```

```
CASE_SCAN:
```

```

/*  BEING PASSED THE CASE LABEL WE SCAN FIXING LABELS AS WE
    FIND THEM UNTIL FINDING END-PERFORM.  THE GO TO
    DEPENDING ON IS THEN SET UP.  */

```

```
PROCEDURE(OLD_CASE);
```

```
DECLARE OLD_CASE PICTURE '9';
```

```
DECLARE (CASE_PIC,CASE_COUNT) PICTURE '999';
```

```
CASE_COUNT = 1;
```

```
CALL GET_CARD;
```

```
DO WHILE (INDEX(CARD,'END-PERFORM')=0);
```

```
IF INDEX(CARD,CASE_LABEL)>0 THEN DO;
```

```
    PUT FILE(OUT) EDIT('                GO TO Z' || COUNT || ' ')
    (COL(1),A(80));
```

```
    CARD= '                ' || CASE_LABEL || ' ' || CASE_COUNT
    || ' ' || SUBSTR(CARD,INDEX(CARD,'')+2);
```

```
    CASE_COUNT = CASE_COUNT + 1;
```

```
END;
```

```
IF INDEX(CARD,' PERFORM ')>0 THEN
```

```
    CALL PROCESS_PERFORM(COUNT,CASE_NUMBER);
```

```
CALL GET_CARD;
```

```
END;
```

```

PUT FILE(OUT) EDIT('          GO TO Z' || COUNT || ' ')
                    (COL(1),A(80));
COUNT_PLUS = COUNT - 1;
PUT FILE(OUT) EDIT('          Z' || COUNT_PLUS || ' ')
                    (COL(1),A(80));
PUT FILE(OUT) EDIT('          GO TO') (COL(1),A(80));
DO CASE_PIC = 1 TO CASE_COUNT-1;;
  PUT FILE(OUT) EDIT('          ' || CASE_LABEL ||
                    '-' || CASE_PIC || ',') (COL(1),A(80));
END;
PUT FILE(OUT) EDIT('          DEPENDING ON CASE'
                    || OLD_CASE || ' ') (COL(1),A(80));
CARD = '          Z' || COUNT || ' ';
COUNT = COUNT + 1;

END CASE_SCAN;

END MAIN;

```

APPENDIX B
EXAMPLE COBOL PROGRAM

IDENTIFICATION DIVISION.

PROGRAM-ID. PROG6.

AUTHOR. BOB HILB.

INSTALLATION. AU COMPUTER CENTER.

DATE-WRITTEN. NOV 23, 1975.

DATE-COMPILED.

REMARKS.

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. IBM-370-155.

OBJECT-COMPUTER. IBM-370-155.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT STRANGE-FILE ASSIGN TO UT-S-STRANGE.

SELECT PRINT-FILE ASSIGN TO UT-S-PRINTER.

DATA DIVISION.

FILE SECTION.

FD STRANGE-FILE

RECORD CONTAINS 80 CHARACTERS

BLOCK CONTAINS 10 RECORDS

RECORDING MODE IS F

LABEL RECORDS ARE STANDARD

DATA RECORD IS STRANGE-RECORD.

01 STRANGE-RECORD.

02 RECORD-TYPE

PICTURE 9.

02 SUBSCRIPTS.

03 ROW

PICTURE 9(5) USAGE IS COMPUTATIONAL.

03 COLUMNS

PICTURE 9(5) USAGE IS COMPUTATIONAL.

02 TYPE-ONE-RECORD.

03 VALUE-1

PICTURE 9(5) USAGE IS COMPUTATIONAL.

03 FILLER

PICTURE X(67).

02 TYPE-TWO-RECORD

REDEFINES TYPE-ONE-RECORD.

03 VALUE-2

USAGE IS COMPUTATIONAL-1.

03 FILLER

PICTURE X(67).

02 TYPE-THREE-RECORD

REDEFINES TYPE-ONE-RECORD.

03 VALUE-3

USAGE IS COMPUTATIONAL-2.

03 FILLER

PICTURE X(63).

02 TYPE-FOUR-RECORD

REDEFINES TYPE-ONE-RECORD.

03 VALUE-4

PICTURE 9(5) USAGE IS COMPUTATIONAL-3.

03 FILLER

PICTURE X(68).

FD PRINT-FILE

RECORD CONTAINS 132 CHARACTERS

BLOCK CONTAINS 1 RECORDS

RECORDING MODE IS F

LABEL RECORDS ARE STANDARD

DATA RECORD IS PRINT-LINE.

01 PRINT-LINE

PICTURE X(132).

WORKING-STORAGE SECTION.

77 MEAN

PICTURE 9(6)V9(6) VALUE ZEROS.

77 SUMS

PICTURE 9(6)V9(6) VALUE ZEROS.

77 SUM-OF-SQUARES

PICTURE 9(6)V9(6) VALUE ZEROS.

77 VARIANCE

PICTURE 9(6)V9(6) VALUE ZEROS.

```

77 STANDARD-DEVIATION      PICTURE 9(6)V9(6) VALUE ZEROS.
77 I                      PICTURE 9(6).
77 J                      PICTURE 9(6).
77 K                      PICTURE 9(3).
01 STRANGE-MATRIX.
  02 STRANGE-ROW          OCCURS 10 TIMES.
    03 STRANGE-COLUMN    OCCURS 10 TIMES.
      04 STRANGE-ARRAY    USAGE IS COMPUTATIONAL-2.
01 PRINT-FORMAT.
  02 FILLER              PICTURE X      VALUE SPACES.
  02 PRINT-LEADER        PICTURE X(40) VALUE SPACES.
  02 FILLER              PICTURE X(7)  VALUE 'MEAN='.
  02 PRINT-MEAN          PICTURE Z(6).9(6).
  02 FILLER              PICTURE X(3)  VALUE ', '.
  02 FILLER              PICTURE X(15) VALUE 'VARIANCE='.
  02 PRINT-VARIANCE      PICTURE Z(6).9(6).
  02 FILLER              PICTURE X(3)  VALUE ', '.
  02 FILLER              PICTURE X(24)
                        VALUE 'STANDARD DEVIATION='.
  02 PRINT-STANDARD-DEVIATION PICTURE Z(6).9(6).
PROCEDURE DIVISION.
A SECTION.
A100-OPEN-FILES.
  OPEN INPUT STRANGE-FILE
  OUTPUT PRINT-FILE.
A110-INITIALIZE-ARRAY.
  PERFORM A300-ZERO-ARRAY THRU A300-EXIT
  VARYING I FROM 1 BY 1 UNTIL I EQUAL 11
  AFTER J FROM 1 BY 1 UNTIL J EQUAL 11.
A120-READ-STRANGE-FILE.
  READ STRANGE-FILE AT END GO TO A140-COMPUTE-MEAN.
A130-CREATE-ARRAY.
  IF RECORD-TYPE EQUAL '1'
    PERFORM A310-TYPE-1-MOVE THRU A310-EXIT.
  IF RECORD-TYPE EQUAL '2'
    PERFORM A320-TYPE-2-MOVE THRU A320-EXIT.
  IF RECORD-TYPE EQUAL '3'
    PERFORM A330-TYPE-3-MOVE THRU A330-EXIT.
  IF RECORD-TYPE EQUAL '4'
    PERFORM A340-TYPE-4-MOVE THRU A340-EXIT.
  GO TO A120-READ-STRANGE-FILE.
A140-COMPUTE-MEAN.
  PERFORM A350-SUM THRU A350-EXIT
  VARYING I FROM 1 BY 1 UNTIL I EQUAL 11
  AFTER J FROM 1 BY 1 UNTIL J EQUAL 11.
  DIVIDE SUMS BY 100 GIVING MEAN ROUNDED.
A150-VARIANCE-STANDARD-DEVIATE.
  PERFORM A360-SUM-OF-SQUARES THRU A360-EXIT
  VARYING I FROM 1 BY 1 UNTIL I EQUAL 11
  AFTER J FROM 1 BY 1 UNTIL J EQUAL 11.
  COMPUTE VARIANCE = (SUM-OF-SQUARES - SUMS ** 2) / 9900.
  COMPUTE STANDARD-DEVIATION = VARIANCE ** .5.

```

MOVE MEAN TO PRINT-MEAN.
 MOVE VARIANCE TO PRINT-VARIANCE.
 MOVE STANDARD-DEVIATION TO PRINT-STANDARD-DEVIATION.
 MOVE 'THE VALUES FOR THE ORIGINAL ARRAY ARE' TO PRINT-LEADER.
 WRITE PRINT-LINE FROM PRINT-FORMAT
 AFTER POSITIONING 1 LINES.

A165-TEST.

PERFORM A170-COMPUTE-2ND-MEAN BEFORE I EQUAL 1.

PERFORM BEFORE I EQUAL 1.

PERFORM UNTIL I IS GREATER THAN J.

IF I EQUAL 11 THEN
 COMPUTE I = 1

ELSE

IF J EQUAL 11 THEN
 COMPUTE J = 20
 IF K EQUAL 1
 COMPUTE K = 2
 ELSE NEXT SENTENCE

ELSE

IF K EQUAL 5 THEN
 COMPUTE J = 6
 END-IF
 COMPUTE J = 21

PERFORM CASE LABELA(K)
 LABELA(1) COMPUTE I = 99
 LABELA(2)
 COMPUTE I = 98
 END-PERFORM

END-IF
 COMPUTE J = 22

PERFORM VARYING I FROM 1 BY 1
 UNTIL I EQUAL 11
 COMPUTE J = 97
 END-PERFORM

END-IF

END-PERFORM.

PERFORM CASE LABEL(K).
 LABEL(1).
 COMPUTE I = 1.
 LABEL(2).

COMPUTE I = 12.
END-PERFORM.

END-PERFORM.

A170-COMPUTE-2ND-MEAN.
MOVE ZEROS TO SUMS.
PERFORM A350-SUM THRU A350-EXIT
VARYING I FROM 1 BY 1 UNTIL I EQUAL 11
AFTER J FROM 1 BY 1 UNTIL J EQUAL 11.
DIVIDE SUMS BY 100 GIVING MEAN ROUNDED.
A180-2ND-VAR-STANDARD-DEVIATE.
MOVE ZEROS TO SUM-OF-SQUARES.
PERFORM A360-SUM-OF-SQUARES THRU A360-EXIT
VARYING I FROM 1 BY 1 UNTIL I EQUAL 11
AFTER J FROM 1 BY 1 UNTIL J EQUAL 11.
COMPUTE VARIANCE = (SUM-OF-SQUARES - SUMS ** 2) / 9900.
COMPUTE STANDARD-DEVIATION = VARIANCE ** .5.
MOVE MEAN TO PRINT-MEAN.
MOVE VARIANCE TO PRINT-VARIANCE.
MOVE STANDARD-DEVIATION TO PRINT-STANDARD-DEVIATION.
MOVE 'THE VALUES OF THE CONVERTED ARRAY ARE' TO PRINT-LEADER.
WRITE PRINT-LINE FROM PRINT-FORMAT
AFTER POSITIONING 2 LINES.
A190-CLOSE-FILES.
CLOSE STRANGE-FILE
PRINT-FILE.
STOP RUN.
A300-ZERO-ARRAY.
COMPUTE STRANGE-ARRAY (I, J) = 0.
A300-EXIT.
EXIT.
A310-TYPE-1-MOVE.
MOVE VALUE-1 TO STRANGE-ARRAY (ROW, COLUMNS).
A310-EXIT.
EXIT.
A320-TYPE-2-MOVE.
MOVE VALUE-2 TO STRANGE-ARRAY (ROW, COLUMNS).
A320-EXIT.
EXIT.
A330-TYPE-3-MOVE.
MOVE VALUE-3 TO STRANGE-ARRAY (ROW, COLUMNS).
A330-EXIT.
EXIT.
A340-TYPE-4-MOVE.
MOVE VALUE-4 TO STRANGE-ARRAY (ROW, COLUMNS).
A340-EXIT.
EXIT.
A350-SUM.
ADD STRANGE-ARRAY (I, J) TO SUMS.

A350-EXIT.

EXIT.

A360-SUM-OF-SQUARES.

COMPUTE SUM-OF-SQUARES = SUM-OF-SQUARES +
STRANGE-ARRAY (I, J) ** 2.

A360-EXIT.

EXIT.

APPENDIX C
PROCESSED COBOL PROGRAM

IDENTIFICATION DIVISION.

PROGRAM-ID. PROG6.

AUTHOR. BOB HILB.

INSTALLATION. AU COMPUTER CENTER.

DATE-WRITTEN. NOV 23, 1975.

DATE-COMPILED.

REMARKS.

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. IBM-370-155.

OBJECT-COMPUTER. IBM-370-155.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT STRANGE-FILE ASSIGN TO UT-S-STRANGE.

SELECT PRINT-FILE ASSIGN TO UT-S-PRINTER.

DATA DIVISION.

FILE SECTION.

FD STRANGE-FILE

RECORD CONTAINS 80 CHARACTERS

BLOCK CONTAINS 10 RECORDS

RECORDING MODE IS F

LABEL RECORDS ARE STANDARD

DATA RECORD IS STRANGE-RECORD.

01 STRANGE-RECORD.

02 RECORD-TYPE

PICTURE 9.

02 SUBSCRIPTS.

03 ROW

PICTURE 9(5) USAGE IS COMPUTATIONAL.

03 COLUMNS

PICTURE 9(5) USAGE IS COMPUTATIONAL.

02 TYPE-ONE-RECORD.

03 VALUE-1

PICTURE 9(5) USAGE IS COMPUTATIONAL.

03 FILLER

PICTURE X(67).

02 TYPE-TWO-RECORD REDEFINES TYPE-ONE-RECORD.

03 VALUE-2

USAGE IS COMPUTATIONAL-1.

03 FILLER

PICTURE X(67).

02 TYPE-THREE-RECORD REDEFINES TYPE-ONE-RECORD.

03 VALUE-3

USAGE IS COMPUTATIONAL-2.

03 FILLER

PICTURE X(63).

02 TYPE-FOUR-RECORD REDEFINES TYPE-ONE-RECORD.

03 VALUE-4

PICTURE 9(5) USAGE IS COMPUTATIONAL-3.

03 FILLER

PICTURE X(68).

FD PRINT-FILE

RECORD CONTAINS 132 CHARACTERS

BLOCK CONTAINS 1 RECORDS

RECORDING MODE IS F

LABEL RECORDS ARE STANDARD

DATA RECORD IS PRINT-LINE.

01 PRINT-LINE

PICTURE X(132).

WORKING-STORAGE SECTION.

77 CASE1

PICTURE 9(4).

77 CASE2

PICTURE 9(4).

```

77 CASE3          PICTURE 9(4).
77 CASE4          PICTURE 9(4).
77 CASE5          PICTURE 9(4).
77 CASE6          PICTURE 9(4).
77 CASE7          PICTURE 9(4).
77 CASE8          PICTURE 9(4).
77 CASE9          PICTURE 9(4).
77 MEAN           PICTURE 9(6)V9(6) VALUE ZEROS.
77 SUMS           PICTURE 9(6)V9(6) VALUE ZEROS.
77 SUM-OF-SQUARES PICTURE 9(6)V9(6) VALUE ZEROS.
77 VARIANCE       PICTURE 9(6)V9(6) VALUE ZEROS.
77 STANDARD-DEVIATION PICTURE 9(6)V9(6) VALUE ZEROS.
77 I             PICTURE 9(6).
77 J             PICTURE 9(6).
77 K             PICTURE 9(3).
01 STRANGE-MATRIX.
  02 STRANGE-ROW OCCURS 10 TIMES.
    03 STRANGE-COLUMN OCCURS 10 TIMES.
      04 STRANGE-ARRAY USAGE IS COMPUTATIONAL-2.
01 PRINT-FORMAT.
  02 FILLER       PICTURE X VALUE SPACES.
  02 PRINT-LEADER PICTURE X(40) VALUE SPACES.
  02 FILLER       PICTURE X(7) VALUE 'MEAN='.
  02 PRINT-MEAN   PICTURE Z(6).9(6).
  02 FILLER       PICTURE X(3) VALUE ', '.
  02 FILLER       PICTURE X(15) VALUE 'VARIANCE='.
  02 PRINT-VARIANCE PICTURE Z(6).9(6).
  02 FILLER       PICTURE X(3) VALUE ', '.
  02 FILLER       PICTURE X(24)
    VALUE 'STANDARD DEVIATION='.
  02 PRINT-STANDARD-DEVIATION PICTURE Z(6).9(6).
PROCEDURE DIVISION.
A SECTION.
A100-OPEN-FILES.
  OPEN INPUT STRANGE-FILE
  OUTPUT PRINT-FILE.
A110-INITIALIZE-ARRAY.
  PERFORM A300-ZERO-ARRAY THRU A300-EXIT
  VARYING I FROM 1 BY 1 UNTIL I EQUAL 11
  AFTER J FROM 1 BY 1 UNTIL J EQUAL 11.
A120-READ-STRANGE-FILE.
  READ STRANGE-FILE AT END GO TO A140-COMPUTE-MEAN.
A130-CREATE-ARRAY.
  IF RECORD-TYPE EQUAL '1'
    PERFORM A310-TYPE-1-MOVE THRU A310-EXIT.
  IF RECORD-TYPE EQUAL '2'
    PERFORM A320-TYPE-2-MOVE THRU A320-EXIT.
  IF RECORD-TYPE EQUAL '3'
    PERFORM A330-TYPE-3-MOVE THRU A330-EXIT.
  IF RECORD-TYPE EQUAL '4'
    PERFORM A340-TYPE-4-MOVE THRU A340-EXIT.
  GO TO A120-READ-STRANGE-FILE.

```

A140-COMPUTE-MEAN.

PERFORM A350-SUM THRU A350-EXIT
 VARYING I FROM 1 BY 1 UNTIL I EQUAL 11
 AFTER J FROM 1 BY 1 UNTIL J EQUAL 11.
 DIVIDE SUMS BY 100 GIVING MEAN ROUNDED.

A150-VARIANCE-STANDARD-DEVIATE.

PERFORM A360-SUM-OF-SQUARES THRU A360-EXIT
 VARYING I FROM 1 BY 1 UNTIL I EQUAL 11
 AFTER J FROM 1 BY 1 UNTIL J EQUAL 11.
 COMPUTE VARIANCE = (SUM-OF-SQUARES - SUMS ** 2) / 9900.
 COMPUTE STANDARD-DEVIATION = VARIANCE ** .5.
 MOVE MEAN TO PRINT-MEAN.
 MOVE VARIANCE TO PRINT-VARIANCE.
 MOVE STANDARD-DEVIATION TO PRINT-STANDARD-DEVIATION.
 MOVE 'THE VALUES FOR THE ORIGINAL ARRAY ARE' TO PRINT-LEADER.
 WRITE PRINT-LINE FROM PRINT-FORMAT
 AFTER POSITIONING 1 LINES.

A165-TEST.

PERFORM A170-COMPUTE-2ND-MEAN
 PERFORM A170-COMPUTE-2ND-MEAN UNTIL I EQUAL 1.

PERFORM Z100 THRU Z101.
 PERFORM Z100 THRU Z101 UNTIL I EQUAL 1.
 GO TO Z101.

Z100.

PERFORM Z102 THRU Z103 UNTIL I IS GREATER THAN J.
 GO TO Z103.

Z102.

IF I EQUAL 11 THEN
 PERFORM Z500 THRU Z501
 ELSE
 PERFORM Z502 THRU Z503.

GO TO Z103.
 Z103.

COMPUTE CASE1 = (K).
 GO TO Z107.
 GO TO Z108.
 LABEL-001.

COMPUTE I = 1.

GO TO Z108.
 LABEL-002.

COMPUTE I = 12.

GO TO Z108.

Z107.

GO TO

LABEL-001,

LABEL-002,

DEPENDING ON CASE1.

Z108.

GO TO Z101.

Z101.

A170-COMPUTE-2ND-MEAN.

MOVE ZEROS TO SUMS.

PERFORM A350-SUM THRU A350-EXIT

VARYING I FROM 1 BY 1 UNTIL I EQUAL 11

AFTER J FROM 1 BY 1 UNTIL J EQUAL 11.

DIVIDE SUMS BY 100 GIVING MEAN ROUNDED.

A180-2ND-VAR-STANDARD-DEVIATE.

MOVE ZEROS TO SUM-OF-SQUARES.

PERFORM A360-SUM-OF-SQUARES THRU A360-EXIT

VARYING I FROM 1 BY 1 UNTIL I EQUAL 11

AFTER J FROM 1 BY 1 UNTIL J EQUAL 11.

COMPUTE VARIANCE = (SUM-OF-SQUARES - SUMS ** 2) / 9900.

COMPUTE STANDARD-DEVIATION = VARIANCE ** .5.

MOVE MEAN TO PRINT-MEAN.

MOVE VARIANCE TO PRINT-VARIANCE.

MOVE STANDARD-DEVIATION TO PRINT-STANDARD-DEVIATION.

MOVE 'THE VALUES OF THE CONVERTED ARRAY ARE' TO PRINT-LEADER.

WRITE PRINT-LINE FROM PRINT-FORMAT

AFTER POSITIONING 2 LINES.

A190-CLOSE-FILES.

CLOSE STRANGE-FILE

PRINT-FILE.

STOP RUN.

A300-ZERO-ARRAY.

COMPUTE STRANGE-ARRAY (I, J) = 0.

A300-EXIT.

EXIT.

A310-TYPE-1-MOVE.

MOVE VALUE-1 TO STRANGE-ARRAY (ROW, COLUMNS).

A310-EXIT.

EXIT.

A320-TYPE-2-MOVE.

MOVE VALUE-2 TO STRANGE-ARRAY (ROW, COLUMNS).

A320-EXIT.

EXIT.

A330-TYPE-3-MOVE.

MOVE VALUE-3 TO STRANGE-ARRAY (ROW, COLUMNS).

A330-EXIT.

EXIT.

A340-TYPE-4-MOVE.
MOVE VALUE-4 TO STRANGE-ARRAY (ROW, COLUMNS).
A340-EXIT.
EXIT.
A350-SUM.
ADD STRANGE-ARRAY (I, J) TO SUMS.
A350-EXIT.
EXIT.
A360-SUM-OF-SQUARES.
COMPUTE SUM-OF-SQUARES = SUM-OF-SQUARES +
STRANGE-ARRAY (I, J) ** 2.
A360-EXIT.
EXIT.
Z500.

COMPUTE I = 1.

Z501.
Z504.

COMPUTE J = 20
IF K EQUAL 1
COMPUTE K = 2
ELSE NEXT SENTENCE.

Z505.
Z508.

COMPUTE J = 6.

Z509.
Z506.

IF K EQUAL 5 THEN
PERFORM Z508 THRU Z509.

COMPUTE J = 21

COMPUTE CASE1 = (K).
GO TO Z111.

GO TO Z112.
LABELA-001. COMPUTE I = 99

GO TO Z112.
LABELA-002.

COMPUTE I = 98

```
GO TO Z112.
Z111.  GO TO
        LABELA-001,
        LABELA-002,
        DEPENDING ON CASE1.
Z112.

Z507.
Z502.

        IF J EQUAL 11 THEN
PERFORM Z504 THRU Z505
        ELSE
PERFORM Z506 THRU Z507.

        COMPUTE J = 22

PERFORM Z113 THRU Z114 VARYING I FROM 1 BY 1

        UNTIL I EQUAL 11

        GO TO Z114.
Z113.

        COMPUTE J = 97

        GO TO Z114.
Z114.

Z503.
```

APPENDIX D
USER'S MANUAL

The job control language for running the pre-processor
on an IBM 360/370 follows:

```

//*****
//STEP1 EXEC PLIXCLG,PARM='OPT(2)'
//SYSPRINT DD DUMMY
//*      CHANGE DUMMY TO SYSOUT=A FOR A SOURCE LISTING OF THE
//*      FIRST JOB STEP OF THE PRE-PROCESSOR
//SYSIN DD DSNAME=IE212.STEP1,DISP=SHR
//*      IE212.STEP1 IS THE DATA SET CONTAINING THE SOURCE
//*      PROGRAM FOR THE FIRST JOB STEP OF THE PRE-PROCESSOR
//GO.SYSPRINT DD DUMMY
//*      REMOVE THIS CARD TO GET A LISTING OF THE ORIGINAL
//*      COBOL PROGRAM
//GO.SYSIN DD *
//*      PLACE THE COBOL DECK HERE
//GO.ERRORS DD SYSOUT=A      *PRINT OUT ERROR MESSAGES
//GO.OUTPUT DD DSNAME=%%TEMP1,DISP=(NEW,PASS),
//      SPACE=(TRK,(3,3),RLSE),UNIT=DISK,
//      DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//*      TEMPORARY DATA SET TO STORE PROGRAM
//GO.BOTTOM DD DSNAME=%%TEMP2,DISP=(NEW,PASS),
//      SPACE=(TRK,(3,3),RLSE),UNIT=DISK,
//      DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//*      ANOTHER TEMPORARY DATA SET USED BY THE FIRST
//*      JOB STEP
//*****
//STEP2 EXEC PLIXCLG,PARM='OPT(2)'
//SYSPRINT DD DUMMY
//*      CHANGE DUMMY TO SYSOUT=A FOR A SOURCE LISTING OF
//*      THE SECOND JOB STEP OF THE PRE-PROCESSOR
//SYSIN DD DSNAME=IE212.STEP2,DISP=SHR
//*      IE212.STEP2 IS THE DATA SET CONTAINING THE SECOND
//*      JOB STEP OF THE PRE-PROCESSOR
//GO.SYSIN DD DSNAME=%%TEMP1,DISP=(OLD,DELETE)
//      DD DSNAME=%%TEMP2,DISP=(OLD,DELETE)
//*      TAKE THE OUTPUT FROM THE FIRST JOB STEP AS INPUT
//GO.OUT DD DSNAME=%%TEMP3,DISP=(NEW,PASS),
//      SPACE=(TRK,(3,3),RLSE),UNIT=DISK,
//      DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//*      OUTPUT FROM THIS JOB STEP TO BE PASSED TO THE
//*      TO THE NEXT JOB STEP--THE COBOL COMPILER
//GO.ERRORS DD SYSOUT=A      *PRINT OUT ERROR MESSAGES
//*****
//STEP3 EXEC COBUCLG,PARM='**PLACE PARMS DESIRED HERE**'
//SYSUT5 DD DSN=%%UT5,UNIT=SYSDA,SPACE=(CYL,5),DISP=(NEW,PASS)
//COB.SYSIN DD DSNAME=%%TEMP3,DISP=(OLD,DELETE)
//GO.STEPLIB DD DSNAME=SYS1.COBLIB,DISP=SHR
//*      PLACE NECESSARY GO. CARDS HERE

```

The job control language given here will have to be modified

depending on the particular COBOL and PL/I procedures that are available and the requirements that they entail. If it is not desirable to maintain the capability to print out the source listing of the pre-processor, it would be much more efficient to compile and link-edit both job steps and make the whole pre-processor a cataloged procedure with the output options as parameters to the procedure.